Interactive Editing of Voxel-Based Signed Distance Fields

Ole Wegen
Hasso Plattner Institute,
Digital Engineering Faculty,
University of Potsdam, Germany
ole.wegen@hpi.uni-potsdam.de

Jürgen Döllner
Hasso Plattner Institute,
Digital Engineering Faculty,
University of Potsdam, Germany
juergen.doellner@hpi.uni-potsdam.de

Matthias Trapp
Hasso Plattner Institute,
Digital Engineering Faculty,
University of Potsdam, Germany
matthias.trapp@hpi.uni-potsdam.de

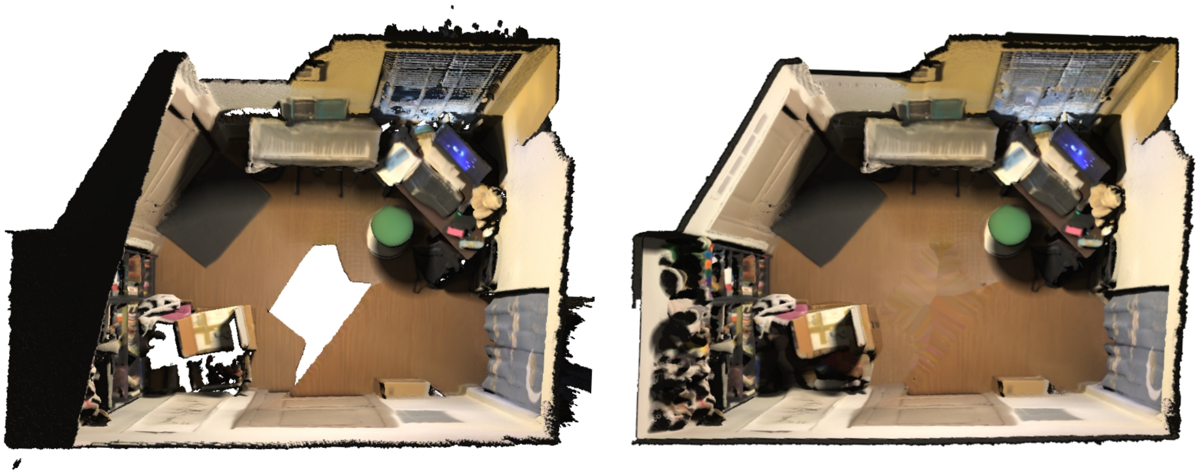


Figure 1: Signed Distance Function (SDF) reconstructed from ScanNet [Dai17] scan. Left image: Original SDF. The ceiling is shaded very dark as the virtual light source is located inside the room. Right image: Cleaned SDF using the implemented manipulation possibilities. The editing time amounted to 5 minutes.

ABSTRACT

Signed distance functions computed in discrete form from given RGB-D data as regular voxel grids can represent manifold shapes as the zero crossing of a trivariate function; the corresponding meshes can be derived by the Marching Cubes algorithm. However, 3D models automatically reconstructed in this way often contain irrelevant objects or artifacts, such as holes or noise, due to erroneous scan data and error-prone reconstruction processes. This paper presents an approach for interactive editing of signed distance functions, derived from RGB-D data in the form of regular voxel grids, that enables the manual refinement and enhancement of reconstructed 3D geometry. To this end, we combine concepts known from constructive solid geometry, where complex models are created from simple base shapes, with the voxel-based representation of geometry reconstructed from real-world scans. Our approach can be implemented entirely on GPU to enable real-time interaction. Further, we present how to implement high-level operators, such as copy, move, and unification.

Keywords: Signed Distance Fields, Interactive Editing, GPU, CUDA

1 INTRODUCTION

Automated 3D reconstruction is a key functionality required in a growing number of application fields, such as robotics, autonomous driving, manufacturing, and spatial digital twins [Kha19]. Volumetric methods for 3D reconstruction are based on computing Signed Distance Functions (SDFs) for regular voxel grids, which encode manifold surfaces as zero-sets of a trivariate implicit function, allowing the acquisition of a large class of objects [Ber02]. Among the most popular raw data formats for volumetric 3D reconstruction methods are colored point clouds or depth-sensitive image data (RGB-D) [Keh14], generated by various scanning and acquisition technologies such as Light Detection and Ranging (LiDAR) sensors, which recently have even become built-in features in mobile devices (e.g., iOS TrueDepth camera); an overview of general 3D reconstruction methods based on RGB-D data is given in [Zol18]. However, “completely digitizing an object or even an entire scene at high-quality is a tedious and time consuming process” [Zol18]. 3D reconstruction results are almost always not *perfect*. For example, a reconstructed 3D model can contain holes if parts of the scene were occluded during scanning, it can show artifacts that result from noise due to reflection in the scan data, or it can contain objects that are irrelevant for the concrete task.

In this paper, we investigate editing techniques that enable the manual refinement of geometry reconstructed based on RGB-D data from real-world indoor and outdoor scenes. To this end, we first convert RGB-D data into a regular voxel-based SDF representation. By combining the voxel-based SDF and procedural shapes into a hybrid scene representation, we enable interactive editing of reconstructed 3D geometry, e.g., for refinement and correction purposes, as shown in Figure 1.

1.1 Problem Statement

The use of SDFs for representing real-world scenes as voxel grids is common since 3D reconstruction by means of volumetric fusion of range data results in such a voxel grid [Cur96]. Furthermore, the creation and manipulation of SDF scenes using Constructive Solid Geometry (CSG) to combine multiple simple shapes into complex objects is also well known; tools such as MagicaCSG enable the creation of SDF scenes in exactly this manner. However, approaches are missing that combine the voxel-grid based SDF representation of real-world scenes reconstructed from scan data with the editing capabilities of CSG, i.e., the combination of procedural geometric shapes by means of set operations [Har95]. With respect to this, the following main challenges have to be considered:

C1: Handling Hybrid Scenes for SDFs: Editing techniques have to handle the hybrid character of SDF scenes, which consist of both procedural primitives (e.g., spheres, boxes) and voxel grids storing the SDF values.

C2: Real-Time Rendering for SDFs: Real-time editing needs interactive frame rates. Rendering voxel-based SDFs requires trilinear interpolation, resulting in many memory reads. If reasonably detailed resolutions for scenes should be achieved and additional (procedural) objects are present in the scene, a Graphics Processing Unit (GPU)-based implementation strategy has to be taken.

1.2 Approach & Contributions

We use a GPU-based approach for creating an SDF from RGB-D scans of real-world objects and scenes. To enable editing and refinement of such scenes at interactive frame rates, we present a new approach that can be implemented entirely on GPUs. It combines the representation of an SDF using a voxel grid with procedural shapes that can be integrated into the grid using set operations. Further, it enables the duplication as well as the translation of scene parts. Additionally, a unification approach enables the merging of the voxel-based and procedural-based SDF representation into a single representation, to increase rendering performance after editing. The presented manipulation techniques could also be transferred to neural geometry representations.

To summarize, this paper presents

1. interaction and manipulation techniques for voxel-based SDFs using procedural shapes,
2. an approach for unification of the scene with subsequent SDF recalculation for persisting changes after editing and thus increasing rendering performance, and
3. a GPU-based implementation of the presented techniques.

The remainder of this work is structured as follows: Section 2 reviews related work with respect to synthesis, rendering, and manipulation of SDF-encoded scenes. Section 3 presents a conceptual overview of the proposed approach. Section 4 details implementation aspects of our Compute Unified Device Architecture (CUDA)-based system. Section 5 evaluates the system's performance and discusses results and limitations. Finally, Section 6 concludes this work and presents ideas for future research.

2 BACKGROUND & RELATED WORK

In the following, we review related work regarding SDF representation, synthesis, rendering, and manipulation.

2.1 SDF Synthesis & Representation

SDFs can be represented by a combination of procedurally defined shapes, in form of voxel volumes, or as weights of a neural network. These representations differ with respect to use-cases and applications.

The procedural representation is mostly used in the context of CSG to build complex objects from simple base shapes. With respect to polygon-based geometry, Willis *et al.* presented PSML (Procedural Shape Modeling Language), which combines shape grammars with sequential statements and can be used to model complex models from 19 simple, predefined base shapes in a hierarchical manner [Wil21]. With respect to SDFs, Reiner *et al.* presented a modeling system that also uses a hierarchical scene graph structure and a CSG-based approach [Rei11]. The advantages of procedural approaches are the low memory consumption and, if visual editing is provided, the user-friendly creation process. The main disadvantage is the tedious nature of the creation process for very complex objects or real-world-based scenes.

The voxel-based representation is commonly used for representing real-world scenes, reconstructed from scan data. Curless and Levoy proposed volumetric fusion for creating such voxel-based SDFs from range images [Cur96]. The range images are fused iteratively into an SDF voxel volume, utilizing the camera extrinsics and intrinsics. Based on that, Izadi *et al.* proposed KinectFusion [Iza11]; camera poses are estimated and utilized for fusing depth images GPU-based in real-time into a global implicit surface model. Such fusion approaches result in Truncated Signed Distance Field (TSDF) volumes that contain distance values only in vicinity to the geometry's surface.
Apart from automatic reconstruction from scan data, voxel-based SDFs can also be created from polygonal meshes or point clouds using distance transform algorithms. An example is the Jump Flooding Algorithm (JFA) [Ron06] that derives a Voronoi tessellation of a voxel grid, with respect to starting seed points. Based on this tessellation, the distance to the nearest seed can be computed per voxel cell. Using the points of a point cloud as starting seeds, the JFA can be used to compute an SDF approximation of the input geometry. Other algorithms utilize hierarchical data structures to directly compute mesh-to-voxel distances, e.g., Mesh-Sweeper [Gue01]. The automatic creation from RGB-D data is one of the main advantages of voxel-based SDF representations. However, the manipulation of such geometry can be challenging due to the fine-grained nature of voxel-grids.

Recent neural approaches store the distance information in the weights of a neural net, which leads to a compact representation with respect to memory consumption [Tak21; Wan21], but increases rendering time, compared to classical representations. Additionally, the editing of such neural representations is challenging and an area of active research.

Our approach combines the procedural and voxel-based representation in order to be able to use automatic reconstruction from RGB-D data together with the easy manipulation known from CSG. The approach can be also adapted to neural representations.

2.2 SDF Rendering

SDFs are typically rendered using ray-marching, where for each pixel of the result image, a ray is emitted into the scene. The ray is advanced, using a fixed step size, until the distance to the surface at the ray's endpoint lies below a certain threshold. Subsequently, the final position of the ray can be used for determining color and normal information for shading.

Hart presented sphere tracing as a faster alternative to the classical ray marching [Har95]. The distance to the surface at a point in space corresponds to the radius of a sphere, in which no geometry is present. Therefore, in each iteration during ray-marching, a ray can be advanced by the distance retrieved from the SDF at the ray's current position, without intersecting geometry. This leads to shorter rendering times compared to classical ray marching. Keinert *et al.* proposed several techniques to enhance sphere tracing, for example an over-relaxation approach for faster tracing [Kei14]. While classical ray-marching can also be used for TSDFs, sphere tracing requires a full SDF. For rendering our scene representation, we rely on sphere tracing, as proposed by Hart.

2.3 SDF Manipulation

In his work on sphere tracing, Hart proved that set operations known from Boolean algebra can be implemented for SDFs using the min/max operators [Har95]. CSG approaches for SDFs, such as the one presented by Reiner [Rei10], utilize this insight for manipulation of procedural-based SDF representations. Zhang presented a GPU-accelerated system for voxel-based SDF modeling, also utilizing these set operations, as well as skeleton-based animation approaches [Zha16]. The focus of his work was, however, on manipulating single objects in rather small voxel grids that could be used, for example, for 3D printing. Our work also utilizes set operations for SDFs, but for real-world-based scenes stored in voxel grids. Additionally, we propose high-level manipulation techniques, such as copy and move functionalities.

3 METHOD

In the following, we give an overview of our SDF creation and rendering process, present our hybrid scene representation, and describe the user interaction concept for SDF scene editing.

3.1 Process Overview

For SDF creation, we iteratively fuse captured RGB-D data into a TSDF volume, as described in [Cur96]. To enable shpere tracing of the reconstructed geometry, we then convert the TSDF into a full SDF. For this we use the JFA, as described in Section 2.1. First, starting seed points are extracted from the TSDF volume by converting to a surface mesh, using Marching Cubes [Lor87], and subsequently sampling the mesh triangles uniformly. Then the JFA is used on these seed points to compute the full SDF. A volume containing color information can be created in the same way.

Rendering the SDF is achieved by means of sphere tracing with additional soft shadow rendering. Subsequently, the SDF can be manipulated, utilizing our heterogeneous SDF representation (Section 1.1, C1).

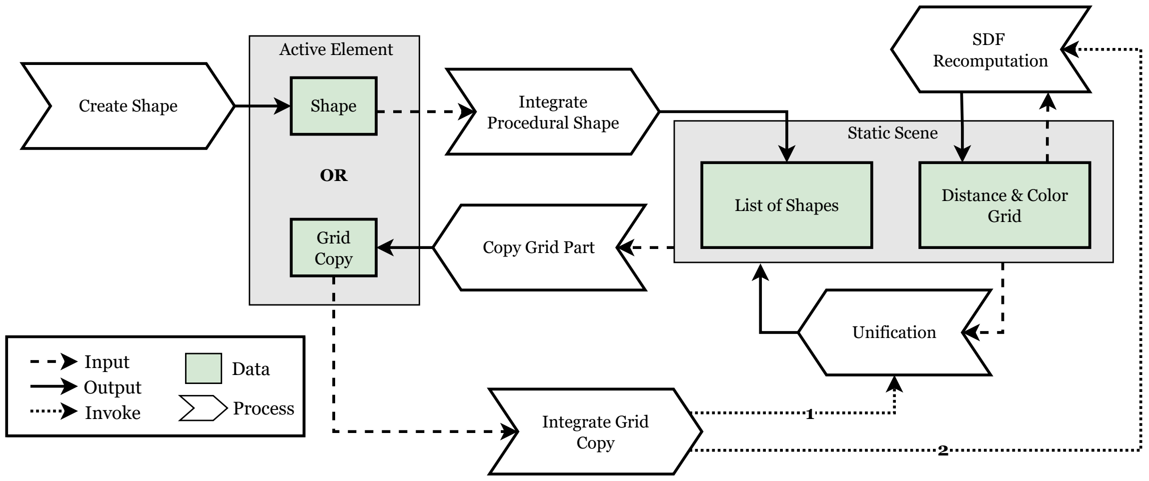


Figure 2: Overview of the processes and data in our approach: Green elements represent data and white arrows represent processes that can be initiated by the user. The static scene consists of two voxel grids, storing distance and color information, as well as a list of shapes that were added to the scene using set operations. The active element can be moved and placed in the scene and is either the currently selected, procedural shape, or a copy of a part of the static scene.

3.2 SDF Scene Representation

The hybrid SDF scene representation, we present, consists of the following components:

Distance & Color Grid: These voxel grids are reconstructed from RGB-D data, as described in Section 3.1.

List of Shapes: A list of procedural shapes that were added to the scene using set operations.

Active Element: The currently selected scene element (shape or voxel grid copy).

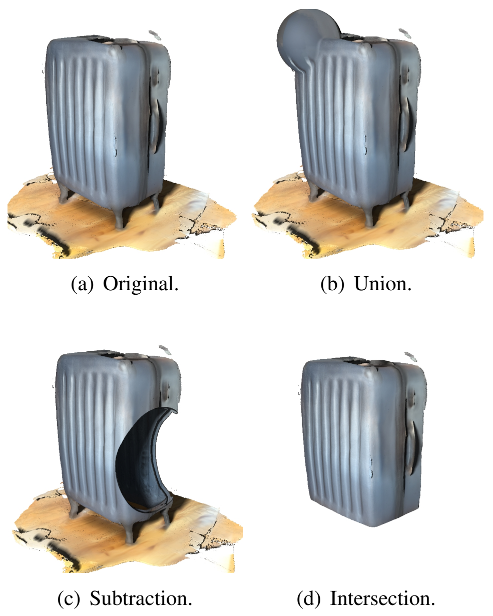


Figure 3: Set operations of an SDF voxel grid (a) with a sphere shape in (b) and (c) and a box shape in (d).

The voxel grids and the list of shapes form the static scene, while the active element represents the dynamic part of the scene that is currently manipulated by the user.

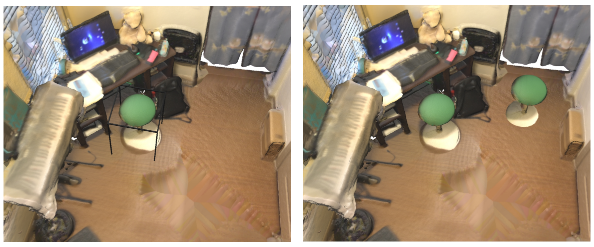
3.3 User Interaction Concept

Figure 2 shows an overview of our system for interactive editing of SDFs. An SDF can be manipulated by creating new, procedural shapes ("Create Shape") and integrating them into the static scene ("Integrate Procedural Shape"). Additionally, parts of the static scene can be copied ("Copy Grid Part") and integrated into the scene at another position ("Integrate Grid Copy"). This copy functionality depends on a unification operation that can also be used for increasing rendering performance. All of the mentioned interaction techniques are described in the following.

Basic Operations. In our approach, geometry is manipulated by geometric shapes (such as spheres or boxes), introduced to a scene. Each shape possesses attributes, such as the position, orientation, size, color, and the scene integration type, all of which can be set by the user, using a Graphical User Interface (GUI). A pointer device (e.g., a mouse) is used to move and place the different shapes in the scene. A once added shape can be selected again by the user to edit its attributes or remove it from the scene.

With respect to the integration type of a shape, three types are supported, corresponding to set operations known from Boolean algebra (Figure 3). They can be implemented for SDFs, using the minimum and maximum operators, as described by Hart [Har95]:

Given two SDFs a and b; an SDF c, resulting from a set operation on the implicit surfaces defined by a



(a) Selecting part of the scene. (b) Placing the copy in the scene.

Figure 4: Example of a copy operation.

and b, can be computed as: $c = \min(a,b)$ (Union), or $\max(-a,b)$ (Subtraction), or $\max(a,b)$ (Intersection).

High-Level Operations. High-level manipulation features include copy and move functionalities, enabling the user to duplicate or move parts of the scene, such as single pieces of furniture in a room. First, the user selects the part of the scene to copy, using a "rubber-band" selection box. The selected part of the static scene is then copied and can be translated and rotated within the scene. Subsequently, the copied part is placed and integrated again into the scene. Figure 4 shows an example for a copy operation In the case of a move operation, a box with the size and position of the selection box is subtracted from the scene after copying.

Unification Operation. With an increasing number of shapes in the scene, the performance of sphere tracing decreases, due to more intricate distance queries. Therefore, the implemented system provides a unification functionality to convert a scene, consisting of a voxel grid and a list of procedural shapes, into a unified voxel grid to improve rendering performance (Section 1.1, C2). This functionality is also used when a copied part of the scene should be integrated back into the scene. After unification, the list of shapes is cleared and the new, unified voxel grid replaces the old one.

4 IMPLEMENTATION ASPECTS

The SDF creation from RGB-D data (TSDF fusion and JFA) was implemented using Python and C++ together with CUDA kernels for hardware acceleration. The SDF rendering and editing was implemented using C++ and CUDA version 11.3 and will be detailed in the following.

4.1 Scene Rendering

Listing 1 shows example code for the `querySDF` function that is invoked during sphere tracing to retrieve the distance to the surface for any point in space. Instead of only trilinearly interpolating in the voxel grid, the list of added shapes, as well as the active element have to be considered.

```
float querySDF(float3 p, float *voxelGrid, /*...*/) {
 float result = FLT_MAX;
 // Handling voxel grid SDF
 if(bboxHit[0]) { result = queryVoxelGrid(p, voxelGrid); }
 // Handling procedural shapes
 for(int i = 0; i < numberOfShapes; i++) {
  if(bool(bboxHit[i+1])){
   float3 queryPoint = mul(sceneShapes[i].rotation,
                           p-sceneShapes[i].position);
   auto combinationFunc = sdfCombinationFunc[sceneShapes[i].
        integrationType];
   auto shapeFunc = sdfShapeFunc[sceneShapes[i].type];
   result = combinationFunc(result, shapeFunc(queryPoint, sceneShapes
        [i].size));
  }
 }
 // Handling active element
 if(bool(bboxHit[numberOfShapes+1])) {
  float3 queryPoint = mul(activeElement.rotation,
                          p-activeElement.position);
  auto combinationFunc = sdfCombinationFunc[activeElement.
        integrationType];
  result = combinationFunc(result, activeElementSdf(queryPoint,
        activeElement));
 }
 return result;
}
```

Listing 1: CUDA code for querying the distance in an SDF scene.

Before sphere tracing, for each ray, the intersection of the ray with the bounding box of each shape and the voxel grid is tested. The results of these bounding box tests are stored into an array (`bboxHit`). This array can then be used to skip all shapes that the ray cannot hit. Apart from that, the following variables and functions are used in the code:

`sceneShapes` is the array of shapes that were already added to the scene.

`activeElement` is the currently selected shape or grid copy.

`sdfCombinationFunc` is an array of functions, implementing different set operations.

`sdfShapeFunc` is an array of signed distance functions for different shapes. By indexing into this array (and the `sdfCombinationFunc` array) with the corresponding shape type, unnecessary branching is avoided.

`queryVoxelGrid()` performs trilinear interpolation in a voxel grid.

`mul()` applies a transformation matrix to a point and is used for realizing rotation of objects.

`activeElementSdf()` returns the signed distance for the active element by either calling the corresponding `sdfShapeFunc` in the case of a procedural shape or `queryVoxelGrid` in the case of a voxel grid copy.

After a ray has terminated, the ray's endpoint is used to retrieve the surface color from the color volume. Additionally, synthetic soft shadows can be rendered by including an additional tracing step from the surface point to a light source to check for occluding geometry in between.

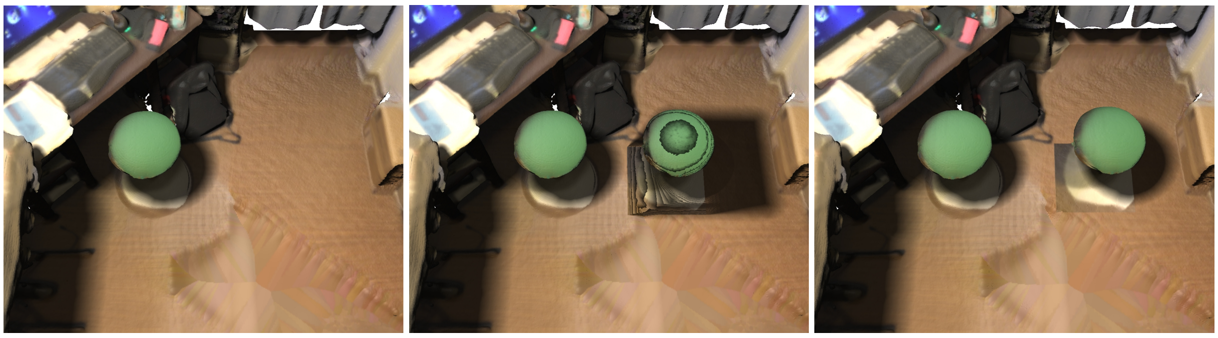


Figure 5: Copying parts of the input scene (a) can lead to shadowing artefacts (b). An SDF recalculation step resolves the problem (c).

4.2 Scene Manipulation

As described in Section 3.3, procedural shapes can be added to the scene using set operations implemented by means of min/max operators. Whenever a user adds a shape, it is set as the `activeElement` that can be moved and rotated. On integration (e.g., triggered by clicking the left mouse button), the currently active shape gets appended to the `sceneShapes` array. During scene rendering, for each ray, the shape ID of the shape with the minimal distance to the ray's endpoint is stored. This allows for shape selection by point-and-click, as each pixel of the rendered frame can be matched to the corresponding shape by means of the stored shape ID. This ID is also used during shading to retrieve the respective material.

For the copy functionality, the `querySDF` function is used to fill a voxel grid of the size of the selection box with signed distance values. Negative values at border voxels are set to zero to avoid "leaking" at cut borders. This grid copy is then set as the active element. However, for sphere tracing the SDF has to be defined at any point in space, which is not the case for the grid copy, which is only defined within its bounds. This leads to incorrect rendering results. To mitigate this problem, the grid copy therefore returns the distance to its bounding box for points outside this bounding box and the real distances for points inside the bounding box (as suggested by [Rei10]). However, to ensure that the rays during sphere tracing do not stop at the bounding box, the distance to a slightly shrinked bounding box is returned. When the user places the grid copy within the scene, a unification step is performed to integrate the copied geometry.

4.3 Unification & Recalculation

The unification is implemented as a CUDA kernel that executes the `querySDF` function (Listing 1) for each voxel, storing the retrieved distances in a new voxel grid. However, if integrating copied parts into the scene in this manner, problems can arise during shadow computation. As these copied parts return the distance to their bounding box for query points located outside of it, the otherwise invisible bounding boxes result in unwanted shadows and shadowing artefacts (Figure 5(b)). Additionally, the rendering performance can decrease, as a ray first has to approach the bounding box of a copied part and only inside this bounding box can retrieve the actual distances, increasing the number of sphere tracing steps. Further, the number of tracing steps also increases after subtraction and intersection operations, as the computation using minimum and maximum operators only results in a lower bound to the actual distance, as Hart notes in [Har95]. An SDF recalculation step was implemented to mitigate the described problems.

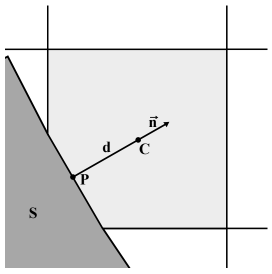


Figure 6: Computation of surface point P.

It first converts the SDF into a voxel grid, containing seed points for a JFA pass. For each voxel that implicitly contains parts of the geometry's surface, a surface point is required. Since, SDFs store only distances not the surface points itself, we compute the surface point as follows (Figure 6). The surface normal $\vec{n}$, which corresponds to the gradient at this point, is computed using central differences. Subsequently, the center point C and the distance d to the surface S are used for computing surface point $P = C - \vec{n} \cdot d$.

After the voxel grid has been filled with seed points, the JFA is executed to obtain the exact distances to the surface for every voxel (see Section 2.1). For the copied parts, the bounding box is omitted before the JFA is applied. Figure 5(c) shows that the shadows are rendered correctly after the recalculation step.

The JFA only results in an approximation of the surface represented by the SDF. To reduce accumulation of errors, when the recalculation is executed several

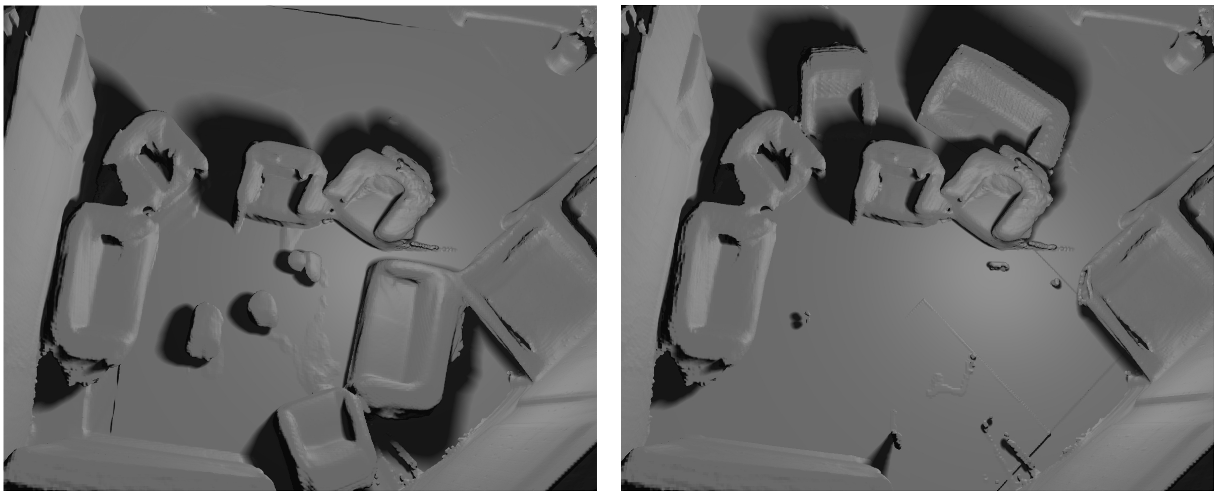

(a) Original SDF from IPad scan (b) SDF after moving objects

Figure 9: Manipulation of an SDF using the move operator. The SDF was reconstructed from an RGB-D scan obtained by an IPad.

times, we use an additional pass at the start of the JFA (the so called 1+JFA variant). Using this JFA variant lead to no observable errors in the geometry, even after repeated SDF recalculation.

5 RESULTS & DISCUSSION

The following section presents exemplary results, created with the implemented manipulation techniques. Subsequently, the run-time performance is evaluated and current limitations are discussed.

5.1 Exemplary Results

Figure 1 (on the first page) shows, how the presented manipulation techniques can be used to refine the 3D geometry of a scene reconstructed from RGB-D data.

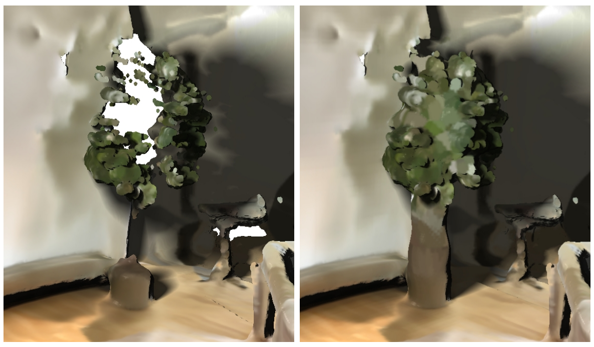

(a) Original (b) Edited

Figure 7: Low-quality SDF of a pot tree, reconstructed from a real-world scan with an IPad. (a) shows the original, noisy, and incomplete SDF. (b) shows the edited SDF, where the tree stump was connected to the tree crown by merging spheres in the scene's color. The editing time amounted to 1-2 minutes.

Holes are filled with shapes in the scene's color, noise in the scene is removed, and unwanted parts, such as the ceiling are cut away. This takes only a few minutes and improves the reconstructed geometry noticeably. Figure 7 shows a similar scenario, where the geometry of a tree, reconstructed from a low-resolution scan, is completed, using the implemented set operations. The common problem of faulty reconstructions from real-world data can therefore be countered, using the proposed manipulation techniques.

Apart from countering problems, such as holes and noise in the scene, the proposed techniques can also be used to add new objects to the scene or alter existing ones, e.g., for artistic purposes. Figure 8 shows an example of this. Additionally, the move functionality can be used for arranging furniture in a reconstructed room anew (Figure 9). This can be useful for interior design and room planning. First, a furnished room

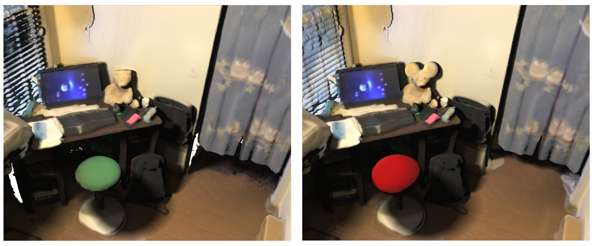

(a) Input SDF (b) Edited SDF

Figure 8: Example of structure and appearance editing. The original 3D reconstruction (a) was edited by filling holes in the scene, changing the color of the stool by merging a red sphere into it's upper part, and adding ears and eyes to the teddy bear by merging spheres of the scene's color (b).

is scanned and reconstructed. Subsequently, new furniture arrangements can be tested, without actually having to move the furniture in the real world. All these manipulation techniques are easy and fast to use, due to the GPU-based implementation and the presented user interaction concept.

5.2 Run-Time Performance Evaluation

In the following, we show an evaluation of the run-time performance of the system. The performance was measured on an AMD Ryzen 5 5600x (6 cores, 3.7 GHz) Central Processing Unit (CPU) with 32 GB DDR4 RAM and an NVIDIA GeForce RTX 3090 GPU with 24 GB VRAM. The application runs on a Windows 10 operating system at a viewport resolution of 1200 × 960 pixels.

For the measurements, reconstructions from a ScanNet scene with different voxel grid resolutions were used, rendered from three different virtual camera views (Figure 10). The rendering time per frame was measured over 12 seconds and the results averaged.

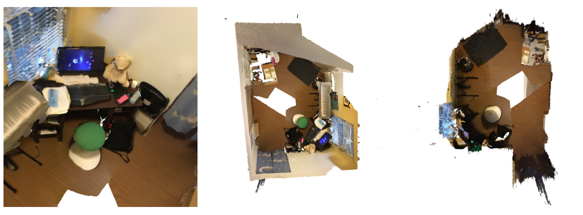
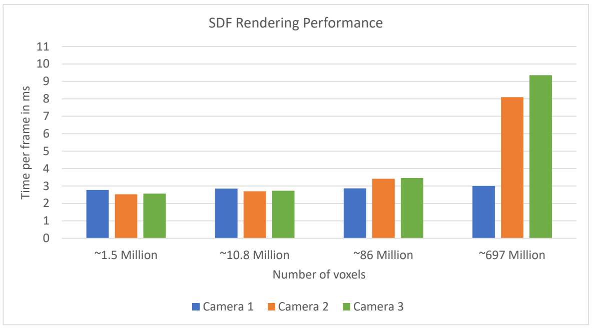

(a) Camera 1 (b) Camera 2 (c) Camera 3

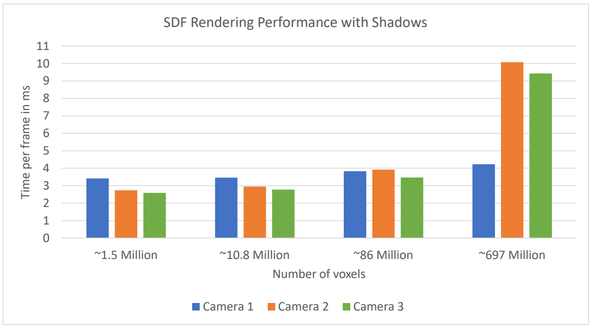
Figure 10: The three different camera configurations used for acquiring performance measurements. Inside the room (a), above (b), and below (c).

Figure 11(a) shows the measurements for rendering the test scene without soft shadows or any additional objects. Even when the full scene is in view (which is the case for camera 2 and 3), it can be rendered in real-time. Figure 11(b) shows measurements for the same set-up, but with soft shadows activated. An increase of up to 2ms rendering time per frame can be observed, especially for the largest scene. Rendering is still possible in real-time.

Figure 12(b) shows how the rendering timings change if additional objects are present in a scene with approx. 697 million voxels. A number of spheres were placed randomly in the scene (Figure 12(a)). The rendering time increases with an increasing amount of objects. Up to 30 objects can be rendered together with the voxel grid at a maximum frame time of around 16ms. For more than 30 objects, the higher frame times result in less than 60 frames per second during rendering. After a unification step is applied, the rendering time always decreases again to approximately the time measured for zero objects in the scene.

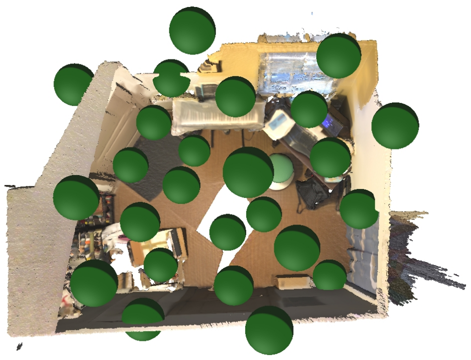


(a) Without shadows

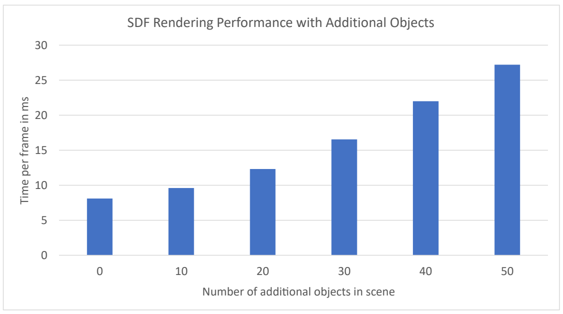


(b) With soft shadows

Figure 11: Performance results for rendering SDFs of different sizes.



(a) Example scene for measuring the performance of the SDF renderer when additional objects are present.



(b) Performance for rendering SDF voxel grids with additional objects added to the scene.

Figure 12: Rendering an SDF voxel grid (917 × 1044 × 728) with additional objects.

Voxel Grid Resolution	**#Voxels**	**Unification**	**SDF Recalculation**
$119 \times 134 \times 95$	$\sim 1.5 \cdot 10^6$	3 ms	24 ms
$229 \times 260 \times 181$	$\sim 10.8 \cdot 10^6$	17 ms	204 ms
$457 \times 520 \times 362$	$\sim 86 \cdot 10^6$	146 ms	1024 ms
$917 \times 1044 \times 728$	$\sim 697 \cdot 10^6$	903 ms	8362 ms

Table 1: Runtime for executing the unification and SDF recalculation steps for different voxel grid resolutions.

Table 1 shows the processing times for the unification and SDF recalculation for voxel grids of different size. These steps can be used for increasing rendering performance, fixing shadow computations after manipulation, and integrating copied or moved parts of the scene. For large voxel grids, the SDF recomputation can require several seconds.

5.3 Limitations

While the feasibility of real-time rendering and editing of real-world scenes represented as SDFs was demonstrated in this work, there are still some open questions and constraints that need to be addressed. A major limitation is currently the memory consumption as full SDF voxel volumes are used. With several hundred million voxels, we are able to represent single rooms with sufficient detail, but larger scenes are still difficult to reconstruct and render. Further, while the unification step increases rendering performance, it is irreversible and makes the subsequent editing of already added shapes impossible. A snapshot-based approach could be used to implement an undo operation. Additionally, while the unification can be executed in less than a second, the SDF recalculation step requires more time. If soft shadows should be rendered, the recalculation step is necessary after every copy/move operation, which can interrupt the work flow if the voxel grid is large, as the execution time of the recalculation step then amounts to several seconds. With regard to user interaction, it is possible to select and edit shapes that were added to the scene using a union operator. Shapes that were integrated into the scene using subtraction or intersection can not be selected for further editing, as during sphere tracing only the shape IDs of shapes where the ray terminates are retrieved. Suitable selection mechanisms have to be developed in the future.

6 CONCLUSIONS & FUTURE WORK

This paper presented an approach for interactive editing of voxel-based signed distance fields. It builds on a hybrid representation consisting of a voxel grid and a number of procedural shapes (Section 1.1, C1), which enables easy manipulation for refining geometry (e.g., closing holes or removing artefacts). Thus, this work is a further building block for creation of high-quality 3D models from real-world scenes, by enabling manual refinement of 3D reconstruction results. High-level manipulation methods, such as copy and move functionality, provide suitable techniques for altering real-world-based geometry and therefore facilitate design and planning processes. The GPU-based implementation of rendering and manipulation enables real-time interaction (Section 1.1, C2).

By exchanging the voxel grid with a neural representation for rendering, the proposed techniques could also be used for altering neural geometry representations. Nevertheless, an additional training phase would be required to transfer the changes back into the neural representation.

The SDF manipulation still has some limitations and extension possibilities. In the future, we will address the problem of memory consumption, by evaluating sparse data structures to reduce memory constraints and allow for larger scenes. Additionally, appearance manipulation (e.g., altering the color volume through a painting technique) would be a useful extension to the already implemented manipulation methods.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work was partially funded by the German Federal Ministry of Education and Research (BMBF) through grants 01IS18092 ("md-ViPro") and 01IS19006 ("KI-LAB-ITSE").

REFERENCES

[Ber02] Fausto Bernardini and Holly Rushmeier. "The 3D Model Acquisition Pipeline". In: *Computer Graphics Forum* 21.2 (2002), pp. 149–172. DOI: https://doi.org/10.1111/1467-8659.00574. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/1467-8659.00574.

[Cur96] Brian Curless and Marc Levoy. "A Volumetric Method for Building Complex Models from Range Images". In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques.* SIGGRAPH '96. New York, NY, USA: Association for Computing Machinery, 1996, pp. 303–312. ISBN: 0897917464. DOI: 10.1145/237170.237269.

[Dai17] A. Dai et al. “ScanNet: Richly-Annotated 3D Reconstructions of Indoor Scenes”. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, July 2017, pp. 2432–2443. DOI: 10.1109/CVPR.2017.261.

[Gue01] A. Gueziec. “"Meshsweeper": dynamic point-to-polygonal mesh distance and applications”. In: *IEEE Transactions on Visualization and Computer Graphics* 7.1 (2001), pp. 47–61. DOI: 10.1109/2945.910820.

[Har95] John Hart. “Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces”. In: *The Visual Computer* 12 (June 1995), pp. 527–545. DOI: 10.1007/s003710050084.

[Iza11] Shahram Izadi et al. “KinectFusion: real-time 3D reconstruction and interaction using a moving depth camera”. In: *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*. Ed. by Jeffrey S. Pierce, Maneesh Agrawala, and Scott R. Klemmer. ACM, 2011, pp. 559–568. DOI: 10.1145/2047196.2047270.

[Keh14] Wadim Kehl, Nassir Navab, and Slobodan Ilic. “Coloured signed distance fields for full 3D object reconstruction”. In: *British Machine Vision Conference, BMVC 2014, Nottingham, UK, September 1-5, 2014*. Ed. by Michel François Valstar, Andrew P. French, and Tony P. Pridmore. BMVA Press, 2014.

[Kei14] Benjamin Keinert et al. “Enhanced Sphere Tracing”. In: *Smart Tools and Apps for Graphics - Eurographics Italian Chapter Conference*. Ed. by Andrea Giachetti. The Eurographics Association, 2014. ISBN: 978-3-905674-72-9. DOI: 10.2312/stag.20141233.

[Kha19] Siavash H. Khajavi et al. “Digital Twin: Vision, Benefits, Boundaries, and Creation for Buildings”. In: *IEEE Access* 7 (2019), pp. 147406–147419. DOI: 10.1109/ACCESS.2019.2946515.

[Lor87] William E. Lorensen and Harvey E. Cline. “Marching cubes: A high resolution 3D surface construction algorithm”. In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1987, Anaheim, California, USA, July 27-31, 1987*. Ed. by Maureen C. Stone. ACM, 1987, pp. 163–169. DOI: 10.1145/37401.37422.

[Rei10] Tim-Christopher Reiner. “Interactive Modeling with Distance Fields”. MA thesis. University of Stuttgart - Institute for Visualization and Interactive Systems, Feb. 2010.

[Rei11] Tim Reiner, Gregor Mückl, and Carsten Dachsbacher. “Interactive modeling of implicit surfaces using a direct visualization approach with signed distance functions”. In: *Computers & Graphics* 35 (June 2011), pp. 596–603. DOI: 10.1016/j.cag.2011.03.010.

[Ron06] Guodong Rong and Tiow-Seng Tan. “Jump Flooding in GPU with Applications to Voronoi Diagram and Distance Transform”. In: *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*. I3D ’06. Redwood City, California: Association for Computing Machinery, 2006, pp. 109–116. ISBN: 159593295X. DOI: 10.1145/1111411.1111431.

[Tak21] Towaki Takikawa et al. “Neural Geometric Level of Detail: Real-Time Rendering With Implicit 3D Shapes”. In: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*. Computer Vision Foundation / IEEE, 2021, pp. 11358–11367.

[Wan21] Yifan Wang, Lukas Rahmann, and Olga Sorkine-Hornung. “Geometry-Consistent Neural Shape Representation with Implicit Displacement Fields”. In: *CoRR* abs/2106.05187 (2021). arXiv: 2106.05187.

[Wil21] Andrew R. Willis et al. “Volumetric procedural models for shape representation”. In: *Graph. Vis. Comput.* 4 (2021), p. 200018. DOI: 10.1016/j.gvc.2021.200018.

[Zha16] Di Zhang. “A GPU Accelerated Signed Distance Voxel Modeling System”. PhD thesis. University of Washington, 2016.

[Zol18] Michael Zollhöfer et al. “State of the Art on 3D Reconstruction with RGB-D Cameras”. In: *Computer Graphics Forum* 37 (May 2018), pp. 625–652. DOI: 10.1111/cgf.13386.